Automatically Synthesizing SQL Queries from Input-Output Examples

Sai Zhang   Yuyin Sun
Computer Science & Engineering
University of Washington, USA
{szhang, sunyuyin}@cs.washington.edu

***Abstract*—Many computer end-users, such as research scientists and business analysts, need to frequently query a database, yet lack enough programming knowledge to write a correct SQL query. To alleviate this problem, we present a *programming by example* technique (and its tool implementation, called SQLSynthesizer) to help end-users automate such query tasks. SQLSynthesizer takes from users an example input and output of how the database should be queried, and then synthesizes a SQL query that reproduces the example output from the example input. If the synthesized SQL query is applied to another, potentially larger, database with a similar schema, the synthesized SQL query produces a corresponding result that is similar to the example output.**

We evaluated SQLSynthesizer on 23 exercises from a classic database textbook and 5 forum questions about writing SQL queries. SQLSynthesizer synthesized correct answers for 15 textbook exercises and all 5 forum questions, and it did so from relatively small examples.

I. Introduction

The big data revolution has resulted in digitization of massive amounts of data. There is a growing population of non-expert database end-users, who need to perform analysis on their databases, but have limited programming knowledge.

Although relational database management systems (RDBMS) and the de facto standard query language (SQL) are perfectly adequate for most end-users' needs, the costs associated with use of database and SQL are non-trivial [15]. The problem is exacerbated by the fact that end-users have diverse backgrounds and include business analysts, commodity traders, finance professionals, and marketing managers. They are not professional programmers. They need to retrieve information from their databases and use the information to support their business decisions. Although most end-users can clearly describe *what* the task is, they are often stuck with the process of *how* to write a correct database query (i.e., a SQL query). Thus, non-expert end-users often need to seek information from online help forums or SQL experts. This process can be laborious and frustrating. Non-expert end-users need a tool that can be used to "describe" their needs and "connect" their intentions to executable SQL queries.

Existing solutions. *Graphical User Interfaces* (GUIs) and *programming languages* are two state-of-the-art approaches to help end-users perform database queries. However, both approaches have limitations.

Many RDBMSes come with a well-designed GUI with lots of features. However, a GUI is often fixed, and does not permit users to personalize a database's functionality for their query tasks. On the other hand, as a GUI supports more and more customization features, users may struggle to discover those features, which can significantly degrade its usability.

Programming languages, such as SQL and other domain-specific query languages (e.g., Java with JDBC), are a fully expressive medium for communicating a user's intention to a database. However, general programming languages have never been easy for end-users who are not professional programmers. Learning a practical programming language (even a simplified domain-specific language, such as MDX [20]) often requires a substantial amount of time and energy that a typical end-user would not prefer to invest.

Our solution: synthesizing SQL queries from input-output examples. This paper presents a technique (and its tool implementation, called SQLSynthesizer) to automatically synthesize SQL queries[1] from input-output examples. Although input-output examples may lead to underspecification, writing them, as opposed to writing declarative specifications or imperative code of any form, is one of the easiest ways for end-users to describe *what* the task is. If the synthesized SQL query is applied to the example input, then it produces the example output; and if the synthesized SQL query is applied to other similar input (potentially much larger database tables), then it produces a corresponding output.

SQLSynthesizer is designed to be used by non-expert database end-users when they do not know how to write a correct SQL query. End-users can use SQLSynthesizer to obtain a SQL query to transform multiple, huge database tables by constructing small, representative input and output example tables. We also envision SQLSynthesizer to be useful in an online education setting (i.e., an online database course). Education initiatives such as Udacity [28] and Coursera [4] are teaming up with experts to provide high-quality online courses to thousands of students worldwide. One challenge, which is not present in a traditional classroom setting, is to provide answers to questions raised by a large number of students. A tool, like SQLSynthesizer, that has the potential of answering SQL query related questions would be useful.

Inferring SQL queries from examples is challenging, primarily for two reasons. First, the standard SQL language is

[1]All queries mentioned in this paper refer to SQL queries that retrieve data from a database but do not update the database.

inherently complex; a SQL query can consist of many parts, such as joins, aggregates, the `GROUP BY` clause, and the `ORDER BY` clause. Searching for a SQL query to satisfy a given example input and output pair, as proved by Sarma et al. [5], is a PSPACE-hard problem. Thus, a brute-force approach such as exhaustively enumerating all syntactically-valid SQL queries and then filtering away those do not satisfy the examples becomes intractable in practice. Second, a SQL query has a rich set of operations: it needs to be evaluated on *multiple* input tables; it needs to perform data grouping, selection, and ordering operations; and it needs to project data on certain columns to form the output. All such operations must be inferred properly and efficiently.

To address these challenges and make example-based SQL query synthesis feasible in practice, SQLSynthesizer focuses on a widely-used SQL subset (Section III) and uses three steps to link a user's intention to a SQL query (Section IV):

- **Skeleton Creation.** SQLSynthesizer scans the given input-output examples and heuristically determines table joins and projection columns in the result query. Then, it creates an incomplete SQL query (called a query skeleton) to capture the basic structure of the result query.
- **Query Completion.** SQLSynthesizer uses a machine learning algorithm to infer a set of accurate and expressive rules, which transforms the input example into the output example. Then, it searches for other missing parts in a query skeleton, and builds a list of candidate queries.
- **Candidate Ranking.** If multiple SQL queries satisfy the given input-output examples, SQLSynthesizer employs the Occam's razor principle to rank simpler queries higher in the output.

Compared to previous approaches [3], [5], [26], [35], SQLSynthesizer has two notable features:

- **It is fully automated.** Besides an example input and output pair, SQLSynthesizer does not require users to provide annotations or hints of any form. This distinguishes our work from competing techniques such as specification-based query inference [35] and query synthesis from imperative code [3].
- **It supports a wide range of SQL queries.** Similar approaches in the literature support a small subset of the SQL language; and most of them can only infer simple select-from-where queries on a single table [3], [5], [26], [35]. By contrast, SQLSynthesizer significantly enriches the supported SQL subset. Besides supporting standard select-from-where queries, SQLSynthesizer also supports many useful SQL features, such as table joins, aggregates (e.g., `MAX`, `MIN`, `SUM`, and `COUNT`), the `GROUP BY` clause, the `ORDER BY` clause, and the `HAVING` clause.

Evaluation. We evaluated SQLSynthesizer's generality and accuracy in two aspects. First, we used SQLSynthesizer to solve 23 SQL exercises from a classic database textbook [23]. We used textbook exercises because they are often designed to cover a wide range of SQL features. Some exercises are even designed on purpose to be challenging and to cover some less realistic, corner cases in using SQL. Second, we evaluated SQLSynthesizer on 5 SQL query related questions collected from popular online help forums, and tested whether SQLSynthesizer can synthesize correct SQL queries for them.

As a result, SQLSynthesizer successfully synthesized queries for 15 out of 23 textbook exercises and all 5 forum problems, within a very small amount of time (9 seconds per exercise or problem, on average). SQLSynthesizer's accuracy and speed make it an attractive tool for end-users to use.

Contributions. This paper makes the following contributions:

- **Technique.** We present a technique that automatically synthesizes SQL queries from input-output examples (Section IV).
- **Implementation.** We implemented our technique in a tool, called SQLSynthesizer (Section V). It is available at: http://sqlsynthesizer.googlecode.com.
- **Evaluation.** We applied SQLSynthesizer to 23 textbook exercises and 5 forum questions. The experimental results show that SQLSynthesizer is useful in synthesizing SQL queries with small examples (Section VI).

II. Illustrating Example

We use an example, described below, to illustrate the use of SQLSynthesizer. The example is taken from a classic database textbook [23] (Chapter 5, Exercise 1) and has been simplified for illustration purposes.[2]

Find the name and the maximum course score of each senior student enrolled in more than 2 courses.

Despite the simplicity of the problem description, writing a correct SQL query can be non-trivial for a typical end-user. An end-user must carefully choose the right SQL features and use them correctly to fulfill the described task.

As an alternative, users can use SQLSynthesizer to obtain the desired query. To use SQLSynthesizer, an end-user only needs to provide it with some small, representative example input and output tables (Figures 1(a) and 1(c)). Then, SQLSynthesizer works in a fully-automatic, push-button way to infer a SQL query that satisfies the given example input and output.

The SQL query, shown in Figure 1(b), first joins two tables on the common `student_id` column, and then groups the joined results by the `student_id` column. Further, the query selects all senior students (using a query condition in the `WHERE` clause) who are enrolled in more than 2 courses (using a condition in the `HAVING` clause). Finally, the query projects the result on the `student.name` column and uses the `MAX` aggregate to compute the maximum course score. To the best of our knowledge, none of the existing techniques [5], [14], [16], [26] can infer this query from the given examples.

III. A SQL Subset Supported in SQLSynthesizer

SQLSynthesizer focuses on a widely-used SQL subset using which a large class of query tasks can be performed. Unfortunately, when designing the SQL subset, we found that no empirical study has ever been conducted to this end, and

[2]This exercise defines 2 tables, whose schemas are shown in Figure 1(a).

student_id	name	level
1	Adam	senior
2	Bob	junior
3	Erin	senior
4	Rob	junior
5	Dan	senior
6	Peter	senior
7	Sai	senior

student_id	course_id	score
1	1	4
1	2	2
2	1	3
2	2	2
2	3	3
3	2	1
4	1	4
4	3	4
5	2	5
5	3	2
5	4	1
6	2	4
6	4	5
7	1	2
7	3	3
7	4	5

$\longrightarrow$

name	max_score
Dan	5
Sai	5

```
SELECT student.name, MAX(enrolled.score)
FROM student, enrolled
WHERE student.student_id = enrolled.student_id
      and student.level = ‘senior’
GROUP BY student.student_id
HAVING COUNT(enrolled.course_id) > 2
```

(a) Two input tables: student (Left) and enrolled (Right) **(b)** A SQL query inferred by SQLSynthesizer **(c)** An output table

Fig. 1. Illustration of how to use SQLSynthesizer to solve the problem in Section II. The user provides SQLSynthesizer with two input tables (shown in (a)) and an output table (shown in (c)). SQLSynthesizer automatically synthesizes a SQL query (shown in (b)) that transforms the two input tables into the output table.

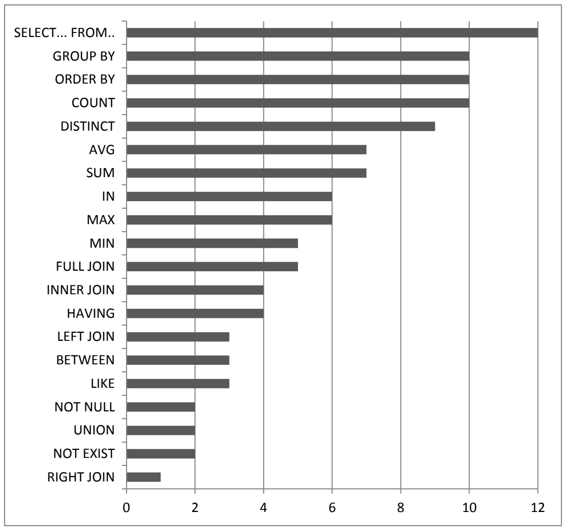


Fig. 2. Survey results of the most widely-used SQL features in writing a database query. Each participant selected the 10 most widely-used SQL features. SQL features with no vote are omitted for brevity.

$$
\begin{aligned}
\langle query \rangle ::= {} & \texttt{SELECT } \langle expr \rangle^{+} \texttt{ FROM } \langle table \rangle^{+} \\
& \texttt{WHERE } \langle cond \rangle^{+} \\
& \texttt{GROUP BY } \langle column \rangle^{+} \texttt{ HAVING } \langle cond \rangle^{+} \\
& \texttt{ORDER BY } \langle column \rangle^{+} \\
\langle table \rangle ::= {} & atom \\
\langle column \rangle ::= {} & \langle table \rangle.atom \\
\langle cond \rangle ::= {} & \langle cond \rangle \texttt{ \&\& } \langle cond \rangle \\
& \mid \langle cond \rangle \texttt{ || } \langle cond \rangle \\
& \mid \texttt{(} \langle cond \rangle \texttt{)} \\
& \mid \langle cexpr \rangle \langle op \rangle \langle cexpr \rangle \\
\langle op \rangle ::= {} & \texttt{=} \mid \texttt{>} \mid \texttt{<} \\
\langle cexpr \rangle ::= {} & const \mid \langle column \rangle \\
\langle expr \rangle ::= {} & \langle cexpr \rangle \mid \texttt{COUNT(}\langle column \rangle\texttt{)} \mid \texttt{COUNT(DISTINCT }\langle column \rangle\texttt{)} \\
& \mid \texttt{MIN(}\langle column \rangle\texttt{)} \mid \texttt{MAX(}\langle column \rangle\texttt{)} \\
& \mid \texttt{SUM(}\langle column \rangle\texttt{)} \mid \texttt{AVG(}\langle column \rangle\texttt{)}
\end{aligned}
$$

Fig. 3. Syntax of the supported SQL subset in SQLSynthesizer: *const* is a constant value and *atom* is a string value, representing a table name or a column name.

little evidence has ever been provided on which SQL features are widely-used in practice. Without such knowledge, deciding which SQL subset to support remains difficult.

To address this challenge and reduce our personal bias in designing the language subset, we first conducted an online survey to ask experienced IT professionals about the most widely-used SQL features in writing database queries (Section III-A). Then, based on the survey results, we designed a SQL subset (Section III-B). Later, we sent the designed SQL subset to the survey participants and conducted a series of follow-up interviews to confirm whether our design would be sufficient in practice.

A. Online Survey: Eliciting Design Requirements

Our online survey consists of 6 questions that can be divided into two parts. The first part includes demographic information including experience in using SQL. In the second part, participants were asked to select the 10 most widely-used SQL features in their minds. Instead of directly asking participants about the SQL features, which might be vague and difficult to respond, we present them a list of standard SQL features in writing a query.

We posted our survey on professional online forums (e.g., StackOverflow) and sent to graduate students at the University of Washington. We received 12 responses. On average, the respondents had 9.5 years of experience in software development (max: 15, min: 5), and 5.5 years of experience in using databases (max: 10, min: 2). In addition, two participants identified themselves as database professionals. Figure 2 summaries the survey results.

B. Language Syntax

Based on the survey results, we designed a subset of the standard SQL language, whose syntax is shown in Figure 3.

```
          Projection column  Aggregate
                  ↓              ↓
SELECT student.name, MAX(enrolled.score)
FROM student, enrolled  ←  Query tables
WHERE student.student_id = enrolled.student_id ← Join conditions
      and student.level = 'senior' ← Query conditions
GROUP BY student.student_id         ←  GROUP BY clause
HAVING COUNT(enrolled.course_id) > 2 ← HAVING clause
ORDER BY student.name  ← ORDER BY clause
```

Fig. 4. An example query using the SQL subset defined in Figure 3.

The supported SQL subset covers all top 10 most widely-used SQL features as voted by the survey participants in Figure 2, except for the `IN` keyword. In addition, the SQL subset supports the `HAVING` keyword since `HAVING` is often used together with the `GROUP BY` clause. Our SQL subset, though by no means complete in writing all possible queries, has significantly enriched the SQL subsets supported by the existing query inference work [5], [26]. Besides being able to write standard select-from-where queries as in [5], [26], our SQL subset also supports table joins, aggregates (e.g., `COUNT`, `MAX`, `MIN`, and `AVG`), the `GROUP BY` clause, the `ORDER BY` clause, and the `HAVING` clause. For readers who are not familiar with the basic SQL idioms, Figure 4 shows an example query using our SQL subset.

When designing the SQL subset, we focused on standard SQL features and excluded user-defined functions and vendor-specific features, such as the `TOP` keyword supported in Microsoft SQLServer. We discarded some standard SQL features, primarily for three reasons. First, some features are designed as syntactic sugar to make a SQL query easier to write; and thus can be safely removed without affecting a language's expressiveness. For example, the `BETWEEN` keyword checks whether a given value is within a specific range or not, and can be simply replaced by two query conditions. Similarly, the `NOT NULL` keyword is also omitted. Second, some features, such as `FULL JOIN`, `LEFT JOIN`, and `RIGHT JOIN`, provide special ways to join tables, and are less likely to be used by non-expert end-users. Third, other features, such as `IN` and `NOT EXIST`, are used to write sub-queries or nested queries, which can significantly increase the complexity in query inference [5]. Related, the `LIKE` keyword is designed for string wildcard matching; determining its matching patterns requires systematic search and can be expensive in practice. Thus, for the sake of inference efficiency, we excluded these keywords.

C. Follow-up Interviews: Feedback about the SQL Subset

After proposing the SQL subset in Figure 3, we performed follow-up email interviews to gain participants' feedback. Participants were first asked to rate the sufficiency of the SQL subset in Figure 3 in writing real-world database queries, on a 6-point scale (5-completely sufficient; 0-not sufficient at all; and in-between values indicating intermediate sufficiency), and then to provide their comments.

Among the 12 received responses, the average rating of the proposed SQL subset is 4.5. Most of the participants rated it 5, or 4. Only one participant rated it 3, because this participant misinterpreted the language syntax and thought it does not support table joins.

Overall, based on the feedback by experienced IT professionals, we believe our SQL subset is usable for end-users in writing common database queries.

IV. Technique

This section first gives an overview of SQLSynthesizer's workflow and high-level algorithm in Section IV-A, and then explains SQLSynthesizer's three steps in details (Section IV-B, Section IV-C, and Section IV-D).

A. Overview

Figure 5 illustrates SQLSynthesizer's workflow. SQLSynthesizer consists of three steps: (1) the "Skeleton Creation" step (Section IV-B) creates a set of query skeletons from the given examples; (2) the "Query Completion" step (Section IV-C) infers the missing parts in each query skeleton, and outputs a list of queries that satisfy the provided example input and output; and (3) the "Candidate Ranking" step (Section IV-D) uses the Occam's razor principle to rank simpler queries higher in the output. Users can inspect SQLSynthesizer's output to select a desired query. If the synthesized SQL queries satisfy the example input and output, but do not address the user's intention, the user can provide SQLSynthesizer with more informative examples and re-apply SQLSynthesizer to the new examples.

Figure 6 sketches SQLSynthesizer's high-level algorithm. Line 2 corresponds to the first "Query Skeleton Creation" step. Lines 3 – 13 correspond to the second "Query Completion" step, in which SQLSynthesizer searches for the query conditions (line 4)[3], aggregates (line 5), and columns in the `ORDER BY` clause (line 6). SQLSynthesizer then builds a list of candidate queries (line 7), and validates their correctness on the examples (lines 8 – 11). Line 14 corresponds to the "Candidate Ranking" step.

B. Skeleton Creation

A query skeleton is an incomplete SQL query that captures the basic structure of the result query. It consists of three parts: query tables, join conditions, and projection columns. To create it, SQLSynthesizer performs a simple scan over the examples, and uses several heuristics to determine each part.

Determining query tables. A typical end-user is often unwilling to provide more than enough example input. Based on this observation, we assume every example input table is used (at least once) in the result query. By default, the query tables are all example input tables. Yet it is possible that one input table will be used multiple times in a query. SQLSynthesizer does not forbid this case; it uses a heuristic to estimate the query tables. If the same column from an input table appears N ($N > 1$) times in the output table, it is highly likely that the input table will be used multiple times in the query, such as being joined with different tables. Thus, SQLSynthesizer replicates the input table N times in the query table set.

[3]Including conditions in the `HAVING` clause, and columns in the `GROUP BY` clause.

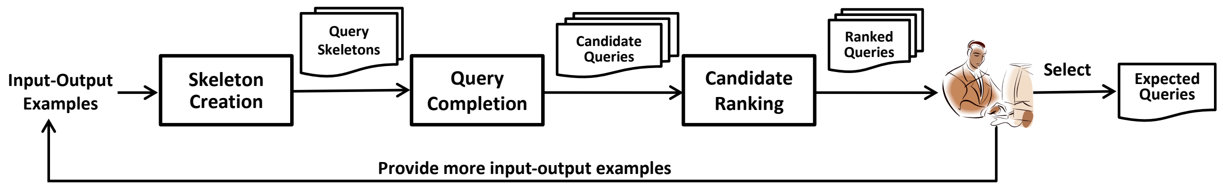


Fig. 5. Illustration of SQLSynthesizer's workflow of synthesizing SQL queries from input-output examples.

```
Input: example input table(s) $T_I$, an example output table $T_O$
Output: a ranked list of SQL queries
 synthesizeSQLQueries($T_I$, $T_o$)
 1: queryList ← an empty list
 2: skeletons ← createQuerySkeletons($T_I$, $T_O$)
 3: for each skeleton in skeletons do
 4:     conds ← inferConditions($T_I$, $T_O$, skeleton)
 5:     aggs ← searchForAggregates($T_I$, $T_O$, skeleton, conds)
 6:     columns ← searchForOrderBys($T_O$, skeleton, aggs)
 7:     queries ← buildQueries(skeleton, conds, aggs, columns)
 8:     for each query in queries do
 9:         if satisfyExamples(query, $T_I$, $T_O$) then
10:             queryList.add(query)
11:         end if
12:     end for
13: end for
14: rankQueries(queryList)
15: return queryList
```

Fig. 6. Algorithm for synthesizing SQL queries from input-output examples.

Determining join conditions. Instead of enumerating all possible ways to join query tables, which may lead to a large number of join conditions, SQLSynthesizer uses two rules (below) to identify the most likely join conditions. SQLSynthesizer repeatedly applies these two rules to join two different tables from the query tables until all tables get joined. First, SQLSynthesizer seeks to join tables on columns with the same name and the same data type. For example, in Figure 1, the `student` table is joined with the `enrolled` table on the `student_id` column, which exists in both tables and has the same data type. If such columns do not exist, SQLSynthesizer uses the second rule to join tables on columns with the same data type (but with different names). For example, suppose the `student_id` column in the `student` table of Figure 1(a) is renamed to `student_key`, SQLSynthesizer will no longer find a column with the same name and data type in table `student` and table `enrolled`. If so, SQLSynthesizer will identify three possible join conditions by only considering columns with the same data type: `student_key = student_id`, `student_key = course_id`, and `student_key = score`; and create three skeletons, each of which uses one join condition.

Determining projection columns. SQLSynthesizer scans each column in the output table, and checks whether a column with the same name exists in an input table. If so, SQLSynthesizer uses the first matched column from the input table as the projection column. Otherwise, the column in the

```
SELECT     student.name, <Aggregate>
FROM       student, enrolled
WHERE      student.student_id = enrolled.student_id
           and  <Query Condition>
GROUP BY   < Column Name(s)>
HAVING     <Query Condition>
ORDER BY   <Column Name(s)>
```

Fig. 7. A query skeleton created for the motivating example in Figure 1. The missing parts (between < and >) will be completed in Section IV-C.

output table must be created by using an aggregate. Take the output table in Figure 1 as an example, SQLSynthesizer determines that the `name` column is from the `student` table and the `max_score` column is created by using an aggregate.

Given an example input and output pair, depending on the number of join conditions, SQLSynthesizer may create multiple skeletons, which share the same query tables and projection columns, but differ in the join condition. For the example in Figure 1, SQLSynthesizer creates one query skeleton shown in Figure 7.

C. Query Completion

In this step, SQLSynthesizer completes the missing parts in each query skeleton and outputs a list of SQL queries that satisfy the given input-output examples.

1) Inferring Query Conditions: SQLSynthesizer casts the problem of *inferring query conditions* as *learning appropriate rules* that can perfectly divide a search space into a positive part and a negative part. In our context, the search space is all result tuples created by joining query tables; the positive part includes all result tuples that contain the output table; and the negative part includes the remaining tuples.

The standard way for rule learning is using a decision-tree-based algorithm. However, a key challenge is how to design a sufficient set of features to capture relevant relations between the example input and output. Existing approaches [26] simply use tuple values in the input table(s) as features, and thus limit their abilities in inferring non-trivial conditions. In particular, merely using tuple values as features can only infer conditions comparing a column value with a constant (e.g., `student.level = 'senior'`), but fails to infer conditions using aggregates (e.g., `COUNT(enrolled.course_id) > 2`), or conditions comparing values of two table columns (e.g., `enrolled.course_id > enrolled.score`). This is primarily because tuple values from the input table(s) do not include enough knowledge about the consequence of applying an aggregate and the potential relations between table columns.

To address this challenge, SQLSynthesizer adds two new types of features to each tuple, and uses them together with

An input table

C1	C2
2	4
2	1
2	1
1	1

Aggregation Features

Group by **C1**						Group by **C2**					
COUNT(**C2**)	COUNT (DISTINCT **C2**)	MIN(**C2**)	MAX(**C2**)	SUM(**C2**)	AVG(**C2**)	COUNT(**C1**)	COUNT (DISTINCT **C1**)	MIN(**C1**)	MAX(**C1**)	SUM(**C1**)	AVG(**C1**)
3	2	1	4	6	2	1	1	2	2	2	2
3	2	1	4	6	2	3	2	1	2	5	5/3
3	2	1	4	6	2	3	2	1	2	5	5/3
1	1	1	1	1	1	3	2	1	2	5	5/3

Comparison Features

C1 = C2	C1 < C2	C1 > C2
0	1	0
0	0	1
0	0	1
1	0	0

Fig. 8. Illustration of two new types of features added by SQLSynthesizer. (Left) An example input table with two columns: C1 and C2. (Center) The aggregation features added by SQLSynthesizer for the input table. (Right) The comparison features added by SQLSynthesizer for the input table.
When grouping all tuples in the input table by column C1, there will be two groups: the first group includes the first 3 tuples with value 2 in the C1 column, and the second group includes the last tuple with value 1 in the C1 column. For example, the first half of the first row in the aggregation feature table shows the results of applying different aggregates to the first group: the number of C2 values is 3 (i.e., COUNT(C2)=3); the number of distinct C2 values is 2 (i.e., COUNT (DISTINCT C2)=2); the minimal C2 value is 1 (i.e., MIN(C2)=1), the maximal C2 value is 4 (i.e., MAX(C2)=4), the sum of C2 values is 6 (i.e., SUM(C2)=6), and the average C2 value is 2 (i.e., AVG(C2)=2).

the existing tuple values for rule learning. The new features explicitly encode useful information about SQL aggregates and table column relations.

- **Aggregation Features**. For each column in the joined table, SQLSynthesizer groups all tuples by *each* tuple's value, and then applies every applicable aggregate[4] to each of the *remaining* columns to compute the corresponding aggregation result. The "Aggregation Features" part in Figure 8 shows an example.
- **Comparison Features**. For each tuple, SQLSynthesizer compares the values of every two type-compatible columns, and records the comparison results (1 or 0) as features. The "Comparison Features" part in Figure 8 shows an example.

SQLSynthesizer employs a variant of the decision tree algorithm, called PART [10], to learn a set of rules as query conditions. We chose PART because it uses a "divide-and-conquer" strategy to build rules incrementally, and thus is faster and consumes less memory than the original decision tree algorithm [22]. Using the new features added by SQLSynthesizer, the PART algorithm is able to discover rules that are hard to identify by only using the original tuple values as features. Figure 9 shows an example.

SQLSynthesizer splits the learned conditions into two disjoint parts, and places each part to the appropriate clause. Specifically, SQLSynthesizer places conditions using aggregates to the HAVING clause, and places other conditions to the WHERE clause. This is based on the SQL language specification: query conditions using aggregates are valid only when they are used together with the GROUP BY clause and are used in the HAVING clause. Take the conditions inferred in Figure 9 as an example, SQLSynthesizer puts the query condition: student.level ='senior' to the WHERE clause, puts condition: COUNT(enrolled.course_id) > 2 to the HAVING clause, and puts column student_id to the GROUP BY clause.

2) Searching for Aggregates: For every column in the output table that has no matched column in the input table(s), SQLSynthesizer searches for the desired aggregate by repeatedly applying each aggregate on every input table column; and checks whether the aggregate produces the same output as in the output table. To speed up the exhaustive search, SQLSynthesizer uses two sound heuristics to filter away infeasible combinations.

- SQLSynthesizer only applies an aggregate to its *type-compatible* table columns. Specifically, the value type of an output column must be compatible with an aggregate's return type. For instance, if an output column contains string values, it cannot be produced by applying the COUNT or COUNT DISTINCT aggregate, or applying the MAX aggregate to a column of integer type. Further, some aggregates have restrictive usages. For example, the AVG and SUM aggregates cannot be applied to columns of string type. SQLSynthesizer encodes such knowledge to prune the search space.
- SQLSynthesizer checks whether each value in the output column exists in the input table. If not, the output column cannot be produced by using the MAX or MIN aggregate.

For the example in Figure 1, SQLSynthesizer determines the max_score column in the output table is produced by using the MAX(score) aggregate.

3) Searching for columns in the ORDER BY clause: SQLSynthesizer scans the values of each column in the output table. If the data values in a column are sorted, SQLSynthesizer appends the column name to the ORDER BY clause.

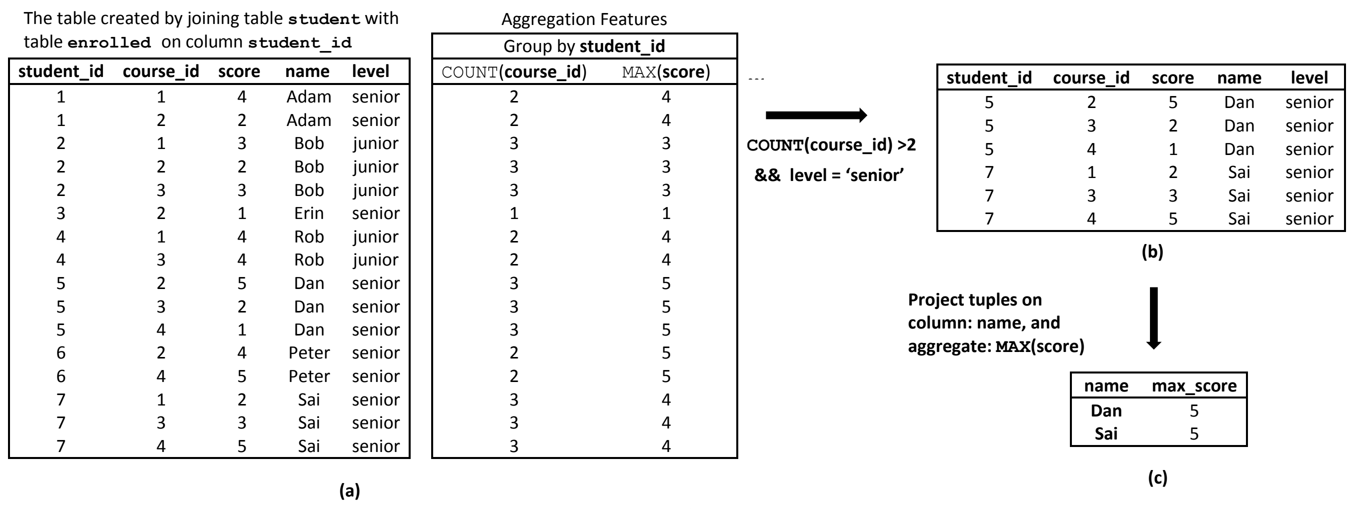
For the output table in Figure 1, SQLSynthesizer determines no column should be added to the ORDER BY clause, since neither output column is sorted.

D. Candidate Ranking

It is possible that multiple SQL queries satisfying the given input-output examples will be returned. SQLSynthesizer employs the Occam's razor principle, which states that the simplest explanation is usually the correct one, to rank simpler queries higher in the output. A simpler query is less likely to overfit the given examples than a complex one, even when both of them satisfy the example input and output.

A SQL query is simpler than another one if it uses fewer query conditions (including conditions in the HAVING and WHERE clauses), or the expressions (including aggregates) in each query condition or clause are pairwise simpler. For example, expression COUNT(student_id) is simpler than COUNT(DISTINCT student_id). Simpler query conditions and expressions often suggest the logics are more common and general.

[4]COUNT, COUNT DISTINCT, MAX, MIN, SUM, and AVG for a column of numeric type ; and COUNT, and COUNT DISTINCT for a column of string type


The table created by joining table **student** with table **enrolled** on column **student_id**

student_id	course_id	score	name	level
1	1	4	Adam	senior
1	2	2	Adam	senior
2	1	3	Bob	junior
2	2	2	Bob	junior
2	3	3	Bob	junior
3	2	1	Erin	senior
4	1	4	Rob	junior
4	3	4	Rob	junior
5	2	5	Dan	senior
5	3	2	Dan	senior
5	4	1	Dan	senior
6	2	4	Peter	senior
6	4	5	Peter	senior
7	1	2	Sai	senior
7	3	3	Sai	senior
7	4	5	Sai	senior

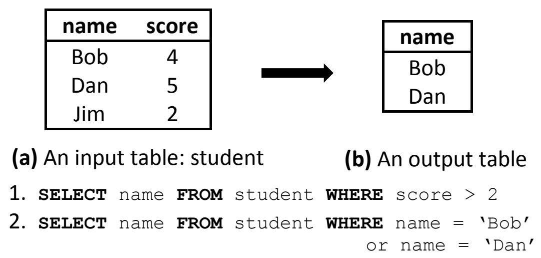
Group by **student_id**	
COUNT(**course_id**)	MAX(**score**)
2	4
2	4
3	3
3	3
3	3
1	1
2	4
2	4
3	5
3	5
3	5
2	5
2	5
3	4
3	4
3	4

...
(a)
COUNT(course_id) >2 && level = 'senior'

student_id	course_id	score	name	level
5	2	5	Dan	senior
5	3	2	Dan	senior
5	4	1	Dan	senior
7	1	2	Sai	senior
7	3	3	Sai	senior
7	4	5	Sai	senior

name	max_score
Dan	5
Sai	5

Fig. 9. Illustration of how the new features added by SQLSynthesizer help in inferring query conditions for the example in Figure 1. (a) shows SQLSynthesizer enriches the original tuple values (Left) with new features (Right). For brevity, only relevant aggregation features are shown. Using the added aggregation features, SQLSynthesizer observes that tuples whose feature values satisfy `COUNT(course_id)>2` and `level='senior'` appear in the output table, and thus infers a query condition that transforms the original table into the table shown in (b), which contains the output table shown in (c). Such query condition is difficult to discover without using the aggregation features. (c) shows the output table, which is produced by projecting the table in (b) on the `name` column and the `MAX(score)` aggregate.



name	score
Bob	4
Dan	5
Jim	2

(a) An input table: student

name
Bob
Dan

(b) An output table

```
1. SELECT name FROM student WHERE score > 2
2. SELECT name FROM student WHERE name = 'Bob'
                                 or name = 'Dan'
```

Fig. 10. Illustration of SQLSynthesizer's query ranking heuristic. SQLSynthesizer produces two queries for the given examples. The first query differs from the second query in using a simpler condition, and thus is ranked higher.

SQLSynthesizer computes a cost for each query, and prefers queries with lower costs. The cost for a SQL query is computed by summarizing the costs of all conditions, aggregates, and expressions appearing in each clause. The cost of each condition, aggregate, and expression is approximated by its length. Figure 10 shows an example.

E. Discussion

Soundness and Completeness. SQLSynthesizer is neither sound nor complete. The primary reason is that several steps (e.g., the Query Skeleton Creation step in Section IV-B) use heuristics to infer query tables, join conditions, and projection columns. Such heuristics are necessary, since they provide a good approximate solution to the problem of finding SQL queries from examples. Although SQLSynthesizer cannot guarantee to infer correct SQL queries for all cases, as demonstrated in Section VI, we find SQLSynthesizer is useful in synthesizing a wide variety of queries in practice.

V. Implementation

We implemented the proposed technique in a tool, called SQLSynthesizer. SQLSynthesizer supports integer and string data types. SQLSynthesizer uses the built-in PART algorithm in the Weka toolkit [13] to learn query conditions (Section IV-C). SQLSynthesizer also uses MySQL [21] as the backend database to validate the correctness of each synthesized SQL query. Specifically, SQLSynthesizer populates the backend database with the given input tables. When a SQL query is synthesized, SQLSynthesizer executes the query on the database to observe whether the query result matches the given output.

VI. Evaluation

We evaluated four aspects of SQLSynthesizer's effectiveness, answering the following research questions:

- What is the success ratio of SQLSynthesizer in synthesizing SQL queries? (Section VI-C1).
- How long does it take for SQLSynthesizer to synthesize a SQL query? (Section VI-C2).
- How much human effort is needed to write sufficient input-output examples for SQL synthesis? (Section VI-C3).
- How does SQLSynthesizer's effectiveness compare to existing SQL query inference techniques? (Section VI-C4).

A. Benchmarks

Our benchmarks are shown in Figure 11.

- We used 23 SQL query related exercises from a classic database textbook [23]. These exercises are from Chapter 5, which systematically introduces the SQL language. We chose textbook exercises because they are designed to cover a wide range of SQL features. Some exercises are even designed on purpose to be challenging and to cover some less realistic, corner cases in using SQL. We used 23 exercises that can be answered using standard SQL language features without any vendor-specific features or user-defined numeric functions. As shown in Figure 11, many textbook exercises involve at least 3 tables. It was unintuitive for us to write the correct query by simply looking at the problem description.
- We searched SQL query related questions raised by real-world database users from 3 popular online forums [6],

Benchmarks			SQLSynthesizer					Query by
ID	Source	#Input Tables	Example Size	Rank	Time Cost (**s**)	Cost in Writing Examples (**m**)	#Iterations	Output [26]
1	Textbook Ex 5.1.1	4	28	1	8	4	2	**Y**
2	Textbook Ex 5.1.2	4	35	**X**	4	10	10	**N**
3	Textbook Ex 5.1.3	2	52	1	4	6	3	**N**
4	Textbook Ex 5.1.4	3	49	**X**	4	10	10	**N**
5	Textbook Ex 5.1.5	2	17	1	3	3	1	**N**
6	Textbook Ex 5.1.6	3	44	1	2	6	3	**N**
7	Textbook Ex 5.1.7	1	10	1	2	1	1	**N**
8	Textbook Ex 5.1.8	1	9	1	2	1	1	**N**
9	Textbook Ex 5.1.9	2	22	1	4	2	5	**N**
10	Textbook Ex 5.1.10	2	19	1	2	3	3	**Y**
11	Textbook Ex 5.1.11	4	22	**X**	3	10	10	**N**
12	Textbook Ex 5.1.12	4	14	**X**	1	10	10	**N**
13	Textbook Ex 5.2.1	2	12	1	3	2	1	**N**
14	Textbook Ex 5.2.2	3	18	1	3	4	2	**N**
15	Textbook Ex 5.2.3	3	21	**X**	4	4	4	**N**
16	Textbook Ex 5.2.4	3	28	1	3	6	3	**N**
17	Textbook Ex 5.2.5	3	24	**X**	2	5	1	**N**
18	Textbook Ex 5.2.6	3	19	1	3	6	3	**N**
19	Textbook Ex 5.2.7	3	25	1	3	5	3	**N**
20	Textbook Ex 5.2.8	3	38	**X**	3	10	10	**N**
21	Textbook Ex 5.2.9	3	31	1	2	7	3	**N**
22	Textbook Ex 5.2.10	3	35	1	5	5	3	**N**
23	Textbook Ex 5.2.11	3	30	**X**	3	10	10	**N**
24	Forum Question 1	1	11	1	3	2	2	**N**
25	Forum Question 2	1	8	1	2	2	2	**N**
26	Forum Question 3	1	24	1	1	3	2	**N**
27	Forum Question 4	4	13	1	120	1	1	**N**
28	Forum Question 5	3	10	1	1	2	2	**N**

Fig. 11. Experimental results. Column "Benchmarks" describes the characteristics of our benchmarks. Column "#Input Tables" shows the number of input tables used in each benchmark. Column "SQLSynthesizer" shows SQLSynthesizer's results. Column "Example Size" shows the number of tuples (i.e., rows) in all user-written example input and output tables. Column "Rank" shows the absolute rank of the correct SQL query in SQLSynthesizer's output. "**X**" means SQLSynthesizer fails to produce a correct answer. Column "Time Cost (s)" shows the total time (in seconds) used in synthesizing queries by SQLSynthesizer. Column "Cost in Writing Examples (m)" shows the total time (in minutes) used in writing examples by the SQLSynthesizer user. Column "#Iterations" shows the number of interactive rounds in using SQLSynthesizer. Column "Query by Output" shows the results of the existing *Query by Output* (QBO) [26] technique. In this column, "**Y**" means QBO produces the correct SQL query, and "**N**" means QBO fails to produce the correct query. The time cost of QBO is comparable to SQLSynthesizer, and is omitted for brevity.

[25], [27]. We focused on questions about how to write queries using standard SQL features. We excluded questions that were vaguely described or were obviously wrong, and discarded questions that had been proved to be unsolvable by using SQL (e.g., computing a transitive closure). We collected 5 non-trivial forum questions (all questions are available at: [9]), among which two questions even did not receive any reply on the forum. Writing a good forum post is often harder than seeking information from internet or asking a SQL expert (since the post has to clearly describe the problem), and these end-users had already tried but failed to find the correct SQL query before they wrote the post.

B. Evaluation Procedure

We used SQLSynthesizer to solve each textbook exercise and forum question. If an exercise or problem was associated with example input and output, we directly applied SQLSynthesizer on those examples. Otherwise, we manually wrote some examples. All examples are written by the second author of this paper, who is not SQLSynthesizer's major developer.

In the evaluation, the SQLSynthesizer user wrote examples in plain text files using CSV format. SQLSynthesizer parses the files, and automatically infers the data type of each column based on the provided values.

We checked SQLSynthesizer's correctness by comparing its output with the expected SQL queries. For textbook exercises, we compared SQLSynthesizer's output with their correct answers. For forum questions, we first checked SQLSynthesizer's output with the confirmed answer in the same post, if there is any. Otherwise, we manually wrote the correct SQL query and compared it with SQLSynthesizer's output.

For some textbook exercises and forum questions, if SQL-Synthesizer inferred a SQL query that satisfied the input-output examples but did not behave as expected, we manually found an input on which the SQL query mis-behaved and re-applied SQLSynthesizer to the new input. We repeated this process and recorded the total number of interactions.

Our experiments were run on a 2.5GHz Intel Core i5 Mac Mini with 4GB physical memory, running OS X Version 10.8.3.

C. Results

Figure 11 summarizes our experimental results.

1) Success Ratio: SQLSynthesizer synthesized correct SQL queries for 15 out of the 23 textbook exercises and all 5 forum questions. We did not come across any benchmark that can be expressed in our SQL subset, but our algorithm failed to infer the correct query. SQLSynthesizer failed to solve 8 textbook exercises, because these exercises required to write complex existential sub-queries in the join conditions using the `IN` and `NOT EXIST` keywords, which are not currently supported by SQLSynthesizer. Compared to the textbook exercises, the 5 forum questions come from more realistic use cases, and can be expressed in our SQL subset and answered by SQLSynthesizer.

We also observed that our ranking strategy (Section IV-D) was surprisingly effective: for all benchmarks where SQLSynthesizer produced a correct answer, it always ranked the correct SQL query as the first suggestion.

2) Performance: As shown in Figure 11, SQLSynthesizer is very efficient. On average, including benchmarks that SQLSynthesizer failed to produce a correct answer, SQLSynthesizer took less than 8 seconds in total to produce the results (min: 1, max: 120). For 20 benchmarks on which SQLSynthesizer succeeded, the average time cost was only 9 seconds (min: 1, max: 120). Among them, in the worst case, SQLSynthesizer took 120 seconds to synthesize a correct SQL query for a complex forum question, which includes 4 tables and 13 columns. SQLSynthesizer searched over 920 possible queries and validated each of them on the examples before getting the correct answer. Even so, SQLSynthesizer's speed is still acceptable for most practical use-case scenarios.

3) Human Effort: We measured the human effort required to use SQLSynthesizer in two ways. First, the time cost to write sufficient input-output examples. Second, the number of interactive rounds in invoking SQLSynthesizer to synthesize the correct SQL queries.

The human effort required in writing input-output examples is reasonable. For the 20 successful benchmarks, it took the user less than 3.6 minutes on average to write sufficient examples per benchmark (min: 1, max: 7). The average example size is 22 (min: 8, max: 52). To produce the correct SQL query, SQLSynthesizer requires just 2.3 rounds of interaction on average (min: 1, max: 5). For 19 successful benchmarks, the number of interaction rounds is 1 to 3. In general, the user spent slightly more time on benchmarks involving more tables or requiring more complex query conditions.

We also observed that, for 6 of the failed benchmarks, the user gave up after spending 10 minutes adding examples and 10 interaction rounds invoking SQLSynthesizer, since the synthesized SQL queries in each round seemed to get closer to the expected query but still differed in some parts. For the remaining 2 failed benchmarks, the user immediately concluded that SQLSynthesizer could not produce a correct answer after 1 and 4 rounds, respectively; because the synthesized SQL query is significantly different than expected. Such observation is driving us to improve SQLSynthesizer's user experience. For example, as our future work, we plan to enhance SQLSynthesizer to inform users about the solvability of a problem *earlier*.

4) Comparison with an Existing Technique: We compared SQLSynthesizer with *Query By Output* (QBO), a technique to infer SQL queries [26] from query output. We chose QBO because it is the most recent technique and also one of the most accurate SQL query inference techniques in the literature. QBO requires an example input-output pair, and uses the decision tree algorithm [22] to infer a query. However, QBO has three limitations. First, it can only join two tables on their key columns (annotated by users), and requires users to specify how to project the results by annotating the projection columns. Second, it uses the original tuple values from the input tables as learning features, and can only infer simple query conditions. Third, QBO does not support many useful SQL features, such as aggregates, the `GROUP BY` clause, and the `HAVING` clause.

We implemented QBO, annotated each example table as it required, and ran it on the same benchmarks. The results are shown in Figure 11. QBO produced correct answers for only 2 textbook exercises and none of the forum questions. Benchmarks solved by QBO were also solved by SQLSynthesizer. QBO's poor performance is primarily caused by its limited support for learning join conditions, query conditions, and many other SQL features.

We did not compare SQLSynthesizer with other techniques [3], [14]–[16], because these techniques either require different input (e.g., a query log [15], [17] or a snippet of Java code [3]), or produce completely different output (e.g., an excel macro [14], or a text editing script [16]) than SQLSynthesizer. Thus, it is hard to conduct a meaningful comparison.

D. Experimental Discussion

Limitations. The experiments indicate four major limitations of SQLSynthesizer. First, some queries cannot be formulated by our SQL subset due to unsupported features. This limitation is expected. Our future work should address it by including more SQL features in SQLSynthesizer. Second, SQLSynthesizer requires users to provide noise-free input-output examples. Even in the presence of a small amount of noises (e.g., a typo), SQLSynthesizer will declare failure. Third, SQLSynthesizer does not provide any guidance regarding when the user should give up on SQLSynthesizer and assume the SQL cannot be synthesized by SQLSynthesizer. Fourth, SQLSynthesizer assumes the database schema is known by the user. SQLSynthesizer may not be helpful to infer SQL queries for a complex database whose schema is unclear to its users.

Threats to Validity. There are two major threats to validity in our evaluation. First, the 23 textbook exercises and 5 forum questions, though covering a wide range of SQL features, may not be representative enough. Thus, we cannot claim the results can be generalized to an arbitrary scenario. Second, our experiments evaluate SQLSynthesizer's accuracy. A user study is needed to further investigate SQLSynthesizer's usefulness.

Experimental Conclusions. We have three chief findings: **(1)** SQLSynthesizer is effective in synthesizing SQL queries from relatively small examples. **(2)** SQLSynthesizer is fast enough for practical use; and it only needs a small amount of human effort in writing examples; **(3)** SQLSynthesizer produces significantly better results than an existing technique [26].

VII. Related Work

We next discuss two categories of related work on reverse engineering SQL queries and automated program synthesis.

Reverse Engineering SQL Queries. Reverse engineering SQL queries is a technique [5], [26], [35] to improve a database's usability. Zloof's work on *Query by Example* (QBE) [35] provided a high-level query language and a form-based Graphical User Interface (GUI) for writing database queries. To use the QBE system, users need to learn its query language, formulate a query with the language, and fill in the appropriate skeleton tables on the GUI. By contrast, SQLSynthesizer mitigates such learning curves by only requiring users to provide some representative examples to describe their intentions.

Tran et al. [26] proposed a technique, called *Query by Output* (QBO), to infer SQL queries from examples. As we have discussed thoroughly in Section VI-C4, QBO excludes many useful SQL features and can only infer simple select-project-join queries. Our experiments demonstrated that QBO failed to infer queries for many use-case scenarios.

Recently, Sarma et al. [5] studied the *View Definitions Problem* (VDP). VDP aims to find the most succinct and accurate view definition, when the view query is restricted to a specific family of queries. VDP can be solved as a special case in SQLSynthesizer where there is only one input table and one output table. Furthermore, the main contribution of Sarma et al's work is the complexity analysis of three variants of the view definitions problem; there is no tool implementation or empirical studies to evaluate the proposed technique.

Automated Program Synthesis. Program synthesis [11] is a useful technique to create an executable program in some underlying language from specifications that can range from logical declarative specifications to examples or demonstrations [1], [2], [7], [12], [14], [16], [18], [19], [24]. It has been used recently for many applications.

The PADS [8] system takes a large sample of unstructured data and infers a format that describes the data. Related, the Wrangler tool, developed in the HCI community, provides a visual interface for table transformation and data cleaning [16]. These two techniques, though well-suited for tasks like text extraction, cannot be used to synthesize a database query. This is because text extraction tasks use completely different abstractions than a database query task; and the existing tools like PADS and Wrangler lack the support for many database operations such as joins and aggregation.

Harris and Gulwani described a system for learning excel spreadsheet transformation macros from an example input-output pair [14]. Given one input table and one output table, their system can infer an excel macro that filters, duplicates, and/or re-organizes table cells to generate the output table. SQLSynthesizer differs in multiple respects. First, excel macros have significantly different semantics than SQL queries. An excel macro can express a variety of table transformation operations (e.g., table re-shaping), but may not be able to formulate database queries. Second, Harris and Gulwani's approach treats table cells as atomic units, and thus has different expressiveness than SQLSynthesizer. For instance, their technique can generate macros to transform one table to another, but cannot join multiple tables or aggregate query results by certain table columns.

Some recent work proposed query recommendations systems to enhance a database's usability [15], [17]. SQLShare [15] is a web service that allows users to upload their data and SQL queries, permitting other users to compose and reuse. SnipSuggest [17] is a SQL autocompletion system. As a user types a query, SnipSuggest mines existing query logs to recommend relevant clauses or SQL snippets (e.g., the table names for the `FROM` clause) based on the partial query that the user has typed so far. Compared to SQLSynthesizer, both SQLShare and SnipSuggest assume the existence of a query log that contains valuable information. However, such assumption often does not hold for many database end-users in practice. SQLSynthesizer eliminates this assumption and infers SQL queries using user-provided examples.

Cheung et al. [3] presented a technique to infer SQL queries from imperative code. Their technique transforms fragments of application logic (written in an imperative language like Java) into SQL queries. Compared to SQLSynthesizer, their work aims to help developers improve a database application's performance, rather than helping non-expert end-users write correct SQL queries from scratch. If an end-user wishes to use their technique to synthesize a SQL query, she must write a snippet of imperative code to describe the query task. Compared to providing examples, writing correct imperative code is challenging for a typical end-user, and thus may degrade the technique's usability.

VIII. Conclusion and Future Work

This paper studied the problem of automated SQL query synthesis from simple input-output examples, and presented a practical technique (and its tool implementation, called SQLSynthesizer). We have shown that SQLSynthesizer is able to synthesize a variety of SQL queries, and it does so from relatively small examples.

For future work, we plan to conduct a user study to evaluate SQLSynthesizer's usefulness. We are also interested in developing techniques to help users debug their SQL queries by leveraging the recent advance in automated test generation [30], [33] and failure explanation [29], [31], [32], [34].

IX. Acknowledgement

We thank Dan Suciu, Michael Ernst, Yingyi Bu and Hao Lü for their insightful comments on an initial draft. This work was supported in part by NSF grants CCF-1016701 and CCF-0963757.

References

[1] Arvind Arasu, Surajit Chaudhuri, and Raghav Kaushik. Learning string transformations from examples. *Proc. VLDB Endow.*, 2(1):514–525, August 2009.

[2] Davi M.J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *ICFP*, pages 193–204, 2010.

[3] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14, 2013.

[4] Coursera. https://www.coursera.org/.

[5] Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. Synthesizing view definitions from data. In *ICDT*, pages 89–103, 2010.

[6] Database Journal. http://forums.databasejournal.com/.

[7] Kathleen Fisher. LearnPADS: Automatic tool generation from ad hoc data. In *SIGMOD*, 2008.

[8] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. From dirt to shovels: fully automatic tool generation from ad hoc data. In *POPL*, pages 421–434, 2008.

[9] SQLSynthesizer. http://sqlsynthesizer.googlecode.com.

[10] Eibe Frank and Ian H. Witten. Generating accurate rule sets without global optimization. In *ICML*, pages 144–151, 1998.

[11] Sumit Gulwani. Dimensions in program synthesis. In *PPDP*, pages 13–24, 2010.

[12] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, pages 317–330, 2011.

[13] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[14] William R. Harris and Sumit Gulwani. Spreadsheet table transformations from examples. In *PLDI*, pages 317–328, 2011.

[15] Bill Howe, Garret Cole, Nodira Khoussainova, and Leilani Battle. Automatic example queries for ad hoc databases. In *SIGMOD*, pages 1319–1322, 2011.

[16] Sean Kandel, Andreas Paepcke, Joseph Hellerstein, and Jeffrey Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, pages 3363–3372, 2011.

[17] Nodira Khoussainova, YongChul Kwon, Magdalena Balazinska, and Dan Suciu. SnipSuggest: context-aware autocompletion for sql. *Proc. VLDB Endow.*, 4(1):22–33, October 2010.

[18] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. Programming by demonstration using version space algebra. *Mach. Learn.*, 53(1-2):111–156, October 2003.

[19] Tessa A. Lau, Pedro Domingos, and Daniel S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[20] MDX: A query language for OLAP databases. http://msdn.microsoft.com/en-us/library/gg492188.aspx.

[21] MySQL. http://www.mysql.com.

[22] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, March 1986.

[23] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. Addison-Wesley (3rd Edition), 2007.

[24] Rishabh Singh and Sumit Gulwani. Learning semantic string transformations from examples. In *VLDB*, 2012.

[25] StackOverflow. http://www.stackoverflow.com.

[26] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. Query by output. In *SIGMOD*, pages 535–548, 2009.

[27] Tutorialized Forums. http://forums.tutorialized.com.

[28] Udacity. https://www.udacity.com/.

[29] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. Localizing failure-inducing program edits based on spectrum information. In *ICSM*, pages 23–32, 2011.

[30] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. Test generation via dynamic symbolic execution for mutation testing. In *ICSM*, pages 1–10, 2010.

[31] Sai Zhang. Practical semantic test simplification. In *ICSE (NIER track)*, 2013.

[32] Sai Zhang, Yu Lin, Zhongxian Gu, and Jianjun Zhao. Effective identification of failure-inducing changes: a hybrid approach. In *PASTE*, pages 77–83, 2008.

[33] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *ISSTA*, pages 353–363, July 19–21, 2011.

[34] Sai Zhang, Cheng Zhang, and Michael D. Ernst. Automated documentation inference to explain failed tests. In *ASE*, November 2011.

[35] Moshé M. Zloof. Query-by-example: the invocation and definition of tables and forms. In *VLDB*, pages 1–24, 1975.